Tetrahedral Projection using Vertex Shaders

Brian Wylie Patricia Crossno
Lee Ann Fisk Kenneth Moreland
Sandia National Laboratories*

Abstract

Projective methods for volume rendering currently represent the best approach for interactive visualization of unstructured datasets. We present a technique for tetrahedral projection using the programmable *vertex shaders* on current generation commodity graphics cards. The technique is based on Shirley and Tuchman's Projected Tetrahedra (PT) algorithm and allows tetrahedral elements to be volume scan converted within the graphics processing unit. Our technique requires no pre-processing of the data and no additional data structures. Our initial implementation allows interactive viewing of large unstructured datasets on a desktop personal computer.

CR Categories and Subject Descriptors: I.3.8 [Computer Graphics]: Applications; I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling; I.3.3 [Computer Graphics: Picture/Image Generation – *display algorithms*
Additional Keywords: direct volume rendering, PC graphics hardware, volume scan conversion, projection volume rendering

1 INTRODUCTION

In recent years, the capabilities and performance of personal computer (PC) graphics cards have grown dramatically. This growth, driven by the computer gaming and entertainment industry, has opened up new opportunities to the scientific visualization community. Although not specifically designed for scientific visualization, the flexibility of the current generation cards allows them to be utilized in unique and creative ways.

Specifically, our work focuses on programmable *vertex shaders* and their ability to act as Single Instruction Multiple Data (SIMD) machines on incoming vertices. In particular, we are working with nVidia's GeForce family of graphics processors. We use this programability to create a new hardware-supported, three-dimensional geometric primitive, the tetrahedron. Existing primitives are all two-dimensional objects drawn in three-space, such as lines, triangles, quads, and polygons. We provide a three-dimensional object that is rendered volumetrically on a commodity graphics card.

Projective methods for volume rendering of unstructured grids [7][9] work by projecting, in visibility order, the polyhedral cells of a mesh onto the image plane, and incrementally compositing the cell's color and opacity into the final image. There are two main orthogonal components in such techniques: determining a visibility order to be used in projecting the cells [7][13][1][10], and actually computing the contribution of each cell to the image [9][7][6].

In this paper, we present a technique for efficiently computing a cell's contribution to the image. It is orthogonal to the algorithm used for depth sorting of the cells. The application can determine the most suitable sorting algorithm, and once the cells are sorted, they can be sent to the graphics processing unit (GPU). This architecture allows the PC's central processing unit (CPU) to perform the depth sorting, and the GPU to perform the PT algorithm on the tetrahedral primitive. In general, splitting up the work in this fashion allows a more balanced approach to volume rendering. In specific instances the CPU and GPU can be pipelined when computing multiple frames. Our segmented approach also facilitates replacing the visibility sorting algorithm with newer algorithms as they become available.

*Sandia National Laboratories, Albuquerque NM 87185-0822
E-mail {bnwylie | pjcross | lafisk | kmorel}@sandia.gov

2 RELATED WORK

For unstructured grids, the use of projection techniques to accelerate volume rendering has been an active area of research for many years. Projection techniques decompose three-dimensional cells into two-dimensional primitives that can be processed by graphics hardware. Earlier algorithms were designed to take advantage of SGI hardware features. Recently, algorithms have been targeted to run on PC graphics cards. The goal of this research is to achieve realistic volume rendering of unstructured data at interactive rates.

In order to achieve this goal, it is necessary to develop solutions to the two components of projective algorithms, visibility ordering and cell contribution. Starting with Williams's work [13], there has been substantial progress in the development of accurate visibility ordering algorithms [10][1][2]. In fact, the SXMPVO algorithm of Cook et al., is able to sort well over one million cells per second on current hardware.

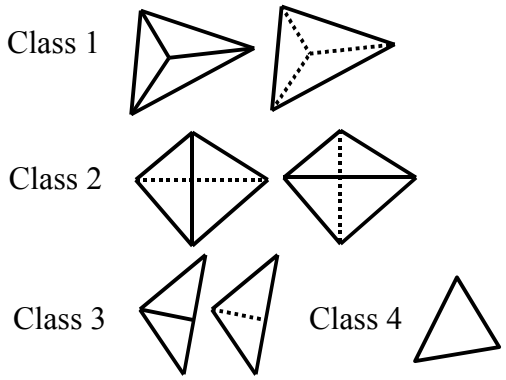

Figure 1: Classification of tetrahedral projections.

The Projected Tetrahedra (PT) algorithm by Shirley and Tuchman [9] demonstrates the potential of using graphics cards to accelerate volume rendering of unstructured data. The PT algorithm splits each of the cells in the grid into tetrahedra. The projected profile of each tetrahedron, with respect to the image plane, is used to classify it and decompose it into a set of triangles. The four classes of projections are shown in Figure 1. The color and opacity values for each triangle vertex are approximated using ray integration through the thickest point of the tetrahedron. The resulting semi-transparent triangles are sorted in depth order, then rendered and composited using graphics hardware. Stein et al. [11] modify the PT algorithm to sort the cells before they are split into tetrahedra. Additionally,

they utilize 2D texture mapping hardware to accelerate opacity interpolation and provide the correct per pixel opacity values to avoid artifacts.

Röttger et al. [8] extend the PT algorithm by correcting the artifacts introduced by the linear interpolation of color and opacity between vertices. They create a three-dimensional texture map to provide hardware support in interpolating along the ray between the front and back faces of a tetrahedral cell. In this texture map, two of the three coordinates correspond to values at the cell entry and exit points, with the third coordinate mapping to the distance through the cell. This texture map is then approximated using two-dimensional texture mapping.

Engel et al. [1] expand on the work of Röttger et al. using the programmable features of the GeForce3 graphics card. To avoid generating additional slices for image quality, they integrate non-linear transfer functions in a pre-processing step. Pre-integrated classification is combined with volume rendering techniques for structured or unstructured cells. However, for this paper, the rendering techniques implemented are for regular grids.

King et al. [5] propose an architecture for tetrahedral scan conversion that would do the visibility sorting in the hardware in addition to the classification, triangulation, and thick vertex calculation. They describe a new hardware buffer, a recirculating fragment buffer or R-buffer that would be used to implement order independent transparency (this is similar to Farias, et al.'s ZSWEEP[4]). To reduce the bandwidth required to send a tetrahedral mesh to the GPU, they propose an extension to the OpenGL API to implement two new primitives, a tetrahedral strip and a tetrahedral fan, and provide 'stripification' algorithms. However, although these ideas are evaluated through simulation, the hardware is not built and the OpenGL extensions do not exist. One of the advantages of both King et al [5] and Farias et al. [4] is that they handle meshes with visibility cycles, while order-dependent techniques would generate incorrect images.

3 APPROACH

Programmable *vertex shaders* do not currently support dynamic vertex creation or topology modification within the vertex program. The topology and number of vertices are fixed and strictly determined by the calling application. Section 3.1 discusses aspects of vertex programming and its effects on execution flow and branching. All of these constraints make the implementation of the PT algorithm impossible if one simply sends the tetrahedral vertices down the graphics pipe. Instead, we need a fixed topology that can be adapted, strictly through vertex manipulations, to represent the two-dimensional projection of a tetrahedron viewed from any angle. We call this fixed topology the ***basis graph***. The *basis graph* is the original tetrahedron with an additional *phantom* vertex. The graph is isomorphic with the two-dimensional projection of the tetrahedron onto the viewing plane, irrespective of viewing angle (see section 3.2). The *basis graph* is sent to the graphics hardware where the vertex program performs all the following steps required by the our algorithm:

1.) Transform to screen space.
2.) Determine projection class.
3.) Calculate the coordinates of the *phantom* vertex.
4.) Map the vertices to the *basis graph*.
5.) Determine depth at thick vertex.
6.) Compute color and opacity for thick vertex.
7.) Multiplex the result to correct output vertex.

The implementation of each of these steps, with respect to the *basis graph*, is covered in detail in the following sections.

3.1 Coding Constraints

Analogous to issues in programming for SIMD architectures, writing vertex programs presents a number of constraints over traditional programming. These include the loss of branching statements for flow control, independent processing of all vertices passed to the GPU with no knowledge about neighboring vertices or connectivity between vertices, and the inability to change execution based on past information. Implementing the PT algorithm under these conditions requires a model where all information about the tetrahedral vertices must be passed in through the input registers, and the same code is redundantly run on each vertex.

These constraints favor universal 'reuse' over algorithmic efficiency. Given the four classes of tetrahedral screen projections shown in Figure 1, clearly not all cases will require identical processing. For instance, although class 1 projections do not need to compute an intersection of the tetrahedral line segments, the lack of flow control leads to performing the intersection calculation anyway. Consequently, we use this result for "thickness" calculations, instead of the more familiar ray-plane intersection test. The emphasis on using the same code for all cases tends to obfuscate what would otherwise be a fairly simple algorithm.

3.2 Basis Graph

The inability to create or delete vertices within a vertex program and the fixed topological constraints of submitted vertices requires the creation of a *basis graph*. The *basis graph* must be able to 'morph' itself into any one of the possible triangle/topology outcomes from the viewpoint dependant projection of the tetrahedral cell.

Our *basis graph* consists of five vertices (four defined by the tetrahedral cell, plus a *phantom* vertex) and a topological specification (see Figure 2). The configuration shown allows us to create any triangulation used by the projections shown in Figure 1 simply by generating new coordinate locations for the five vertices *basis graph*. These coordinates may be computed dynamically based on line intersections (class 2) or we may just swap or copy input vertex locations to accommodate topology or winding constraints.

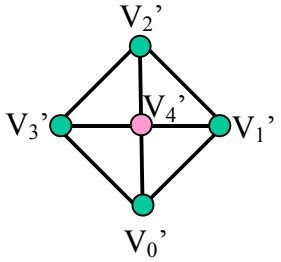

Figure 2: Our *basis graph* with five vertices and their topological specification. Note: the graph illustrates topology and does not represent geometric locations of the vertices.

In Figure 3, we present a pseudo code example of how the *basis graph* is used conceptually. A tetrahedron is simply defined by coordinate vertices $\{V_0, V_1, V_2, V_3\}$. For each tetrahedron we introduce an additional *phantom* vertex, V_P, that is sent down the graphics pipe along with the four *true* vertices. The topology of the *basis graph* shown in Figure 2 can be drawn with the GL_TRIANGLE_FAN primitive. In practice, the input registers for each invocation of the vertex program are loaded with the properties (coordinates, colors, etc…) of all four

tetrahedral vertices and the specific vertex given to the glVertex call is ignored because the vertex program uses at all four vertices in its calculations.

```
depth sort all tetrahedra

for each tetrahedron {V0, V1, V2, V3}
{
    Load {V0, V1, V2, V3} into registers
    Begin TRIANGLE_FAN
      Invoke vertex program for V4'
      Invoke vertex program for V0'
      Invoke vertex program for V1'
      Invoke vertex program for V2'
      Invoke vertex program for V3'
      Invoke vertex program for V0'
    End
}
```

Figure 3: Pseudo code for converting tetrahedral cells with four vertices into the five-vertex *basis graph*.

Using this approach, the modifications to existing application software would be minimal. In fact, by just applying a thin veneer we could support a GL_TETRA_EXT primitive and details like the *phantom* vertex could be abstracted away.

3.3 Classification Determination

Shirley and Tuchman [9] classify the projection of a tetrahedron into four different cases based on the number of triangles required for drawing the projection silhouette from various viewing angles. In each case, one of the vertices of the triangles represents the thickest point in the projection of the tetrahedron. The other vertices of the triangles represent the points where the thickness at the edge of the tetrahedron transitions to zero. The thickness is used in calculating the brightness and opacity of the cell projection. In Class 2, none of the four vertices defining the tetrahedron falls at the thickest point, which is at the intersection of the view vector with the front and back edges, so this case requires the addition of a vertex. Class 1 and 2 are the distinct classes with classes 3 and 4 representing degenerate cases (Figure 1).

Given that we pass the vertex data into the GPU before we know anything about the tetrahedron's projection onto the screen, the input order of the vertices to the vertex program is arbitrary. Therefore, we must deal with every permutation of vertex ordering in the vertex program. We have developed an expanded class table, given in Figure 4, that provides all possible permutations of the relative vertex locations based on input ordering for class 1 and 2.

The vertex program must differentiate between permutations to correctly construct the triangle decomposition with respect to both topology and winding order issues. With permutations there are 14 cases/projections, not including degenerate cases. Since vertices cannot be deleted in a vertex program, we simply create zero area triangles for degenerate cases. The vertex program classifies the tetrahedron into one of the 14 projection cases by conducting a series of geometric tests. The setup terms and the four Boolean tests needed for case determination are listed in Figure 5. Together with the truth table in Table 1, these identify the correct mapping between the two isomorphic graphs. Let **V** = $\{V_0, V_1, V_2, V_3, V_P\}$ be the projected tetrahedron vertices (plus generated *phantom* vertex) and **V'** = $\{V_0', V_1', V_2', V_3', V_4'\}$ be the vertices of the *basis graph*. For each case *i* of the 14 cases, there exists a static onto function f_i: **V** $\rightarrow$ **V'** that maps the projected vertices to the *basis graph*.

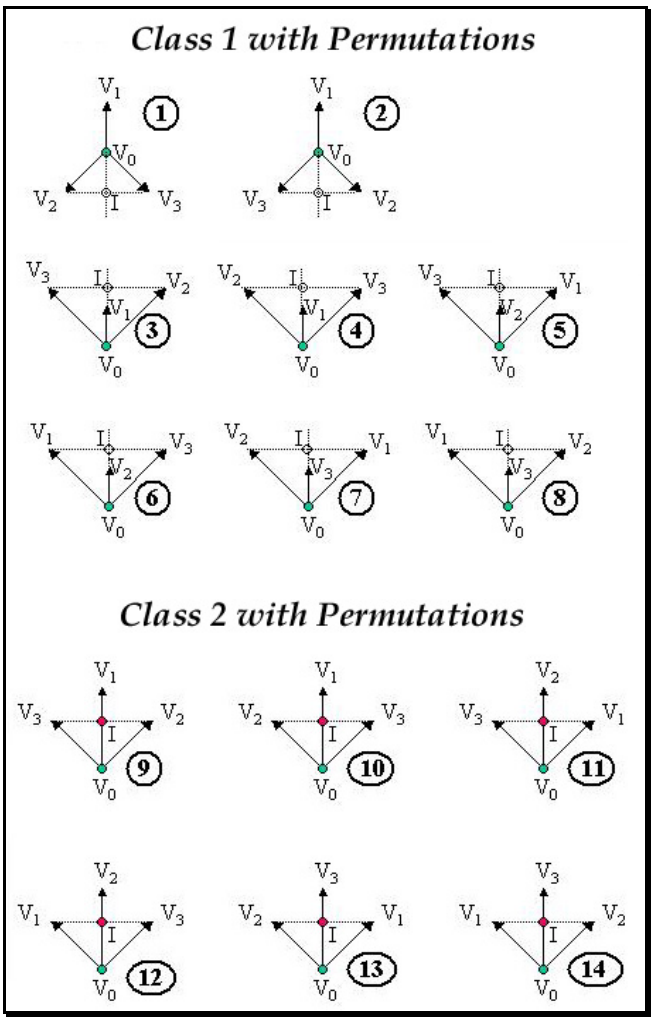

Figure 4: Classifications of projected tetrahedra with permutations (degenerate classes 3 and 4 are not included). The vectors and line segment intersections used in the vertex program are included for illustration.

vec1 = V_1-V_0; **vec2** = V_2-V_0; **vec3** = V_3-V_0;

cross1.z = z component of (**vec1** x **vec2**);
cross2.z = z component of (**vec1** x **vec3**);
cross3.z = z component of (**vec2** x **vec3**);

test1 = (cross1.z * cross2.z < 0);
test2 = (cross1.z * cross3.z > 0);
test3 = (distance V_0 to middle vertex – distance V_0 to **I**) > 0;
test4 = (cross1.z > 0);

Figure 5: Definitions and tests used by the vertex program for classification of projected tetrahedra.

A description of vertex program operation is given below to illustrate the process.

Before the vertex program is run, all tetrahedral vertex properties, $\{V_0, V_1, V_2, V_3\}$, are loaded into the input registers in arbitrary order.

First, the vertex program converts vertices from object space to normalized screen space. The vectors **vec1**, **vec2**, and **vec3** are then computed in screen space. The z-components of the cross products are then computed.

Now we begin an example determination process…

If **test1** is true, then the z terms of **cross1** and **cross2** have different signs. By using the right hand rule this must mean that V_2 and V_3 are on different 'sides' of **vec1** (i.e. cases 1, 2, 3, 4, 9, or 10). Now, if **test2** is true, then cross1.z and cross3.z have the same sign. So, if V_2 is to the right of V_1, then V_3 is also to the right of V_2. Use the same logic for the left side. **Test2** gives us cases (1, 2, 5, 6, 11, or 12). If both **test1** and **test2** are true, that leaves only cases 1 and 2. These only differ in 'winding'. All winding ambiguities are settled using **test4**.

Our example did not discuss the use of test3. Test3 is used to distinguish between Class1[*] and Class2. It simply computes the Euclidean distance from V_0 to the *middle vertex*[†] and the distance from V_0 to the intersection point, **I**. If the distance to the intersection point is less than the distance to the *middle vertex*, than it must be a Class 2 projection. In general, these tests could take many different forms. These particular tests were chosen because they require no additional data structures, no preprocessing, and they can be done in very few vertex program instructions. The complete logic table for the projection case determination is given in Table 1.

Case	Test 1	Test 2	Test 3	Test 4
1	1	1	X	1
2	1	1	X	0
3	1	0	0	0
4	1	0	0	1
5	0	1	0	0
6	0	1	0	0
7	0	0	0	1
8	0	0	0	0
9	1	0	1	0
10	1	0	1	1
11	0	1	1	1
12	0	1	1	0
13	0	0	1	1
14	0	0	1	0

Table 1: The truth table used by the vertex program for projection case determination.

[*] Excluding cases 1 and 2, which have 'Don't Care' logic for test3.

[†] The *middle vertex* is defined as follows: if tests 1 & 2 pass, it's V_0; if test 1 passes and test 2 fails, its V_1; if test 1 fails and test 2 passes, its V_2, and if tests 1 & 2 fail its V_3.

3.4 Isomorphic Mapping

Now that the vertex program has determined the projection case, it must handle the topological and winding issues when mapping the tetrahedral projection onto the *basis graph*. A helpful analogy is to think of the vertex program as multiplexing the input vertices. The program has access to all the vertex coordinates as input (plus the intersection point, which is computed internally). From these it selects one to output.

To demonstrate the isomorphic mapping concept, let us follow the triangle construction for one of our cases. Arbitrarily we have picked case 5. Recall that the first vertex called within a GL_TRIANGLE_FAN primitive is the 'pivot' vertex. The pivot vertex is the 'thick' vertex around which the rest of the 'thin' vertices fan. So for case 5 we know that the proper triangulation is: V_2 as the thick vertex, followed by V_0, V_1, V_3 and back to V_0 to complete the fan[‡]. Figure 6 demonstrates the process our method uses to convert the tetrahedron into screen space triangles.

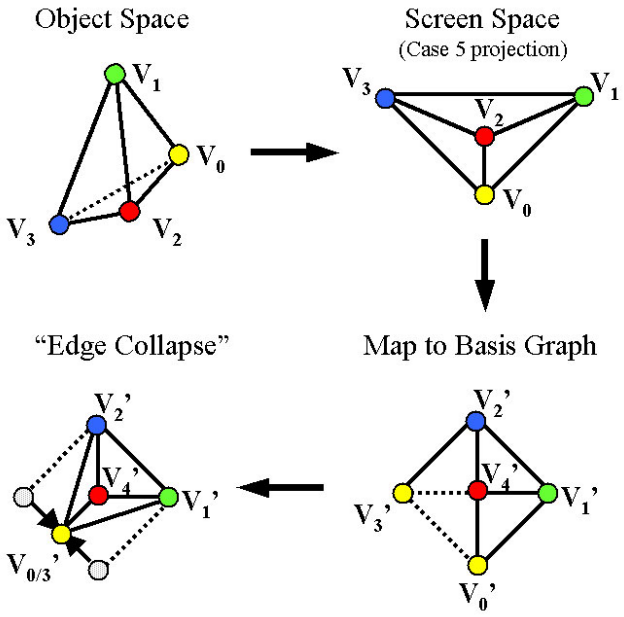

Figure 6: An example showing the vertex to node mapping from a projection instance to the basis graph (shown for a case 5 projection).

Note that it requires 6 vertex calls to construct a fan of 4 triangles, as shown in Figure 3, and that Class 2 projections require 4 triangles. We must always make 6 calls because we do not know a-priori which projection that tetrahedron will make. Since this particular case is a 3-triangle projection, the call to generate the last vertex creates a degenerate triangle.

3.5 Computing Thick Vertex Properties

At this point the vertex program has determined the projection and mapped the tetrahedron onto the *basis graph* to form proper triangles. The only remaining task is to compute the properties of the thick vertex. As discussed in Section 3.1, the logic behind the thick vertex calculations may be unintuitive algorithmically, but code reuse dominated our design criteria.

[‡] The thin vertices may be called in any order as long as the proper winding is adhered to.

The first thing we need to do is compute the thickness ***l*** of the thick vertex. In all cases the coordinates of the intersection point **I** are computed (see Figure 3). The intersection point is determined by intersecting the tetrahedral segments as shown in Figure 3. We compute the thickness, in all cases, by interpolating along these line segments. Class 2 projections are straightforward; we compute an interpolated z1 along one segment and compute an interpolated z2 along the other segment and take the absolute value of the difference. Class 1 projections build upon the Class 2 result by simply multiplying the result by the ratio of the distance from V_0 to the middle vertex over the distance from V_0 to the intersection point (see pseudo code for details).

For linearly varying cells, the color and extinction coefficient calculations are carried out in a similar manner; properties from the vertices are interpolated along the line segments to give the terms C_0, C_1 and τ_1, τ_2. The thick vertex color is then approximated to be $(C_0+C_1)/2$*.

The extinction coefficient $\boldsymbol{\tau}$ is $(\tau_1+\tau_2)/2$. Once we know $\boldsymbol{\tau}$ and the thickness ***l***, we then use the coefficients as lookups into a 2D texture map defined as $1 - \exp(-\tau l)$. This technique gives the correct attenuation of light across a linear varying cell as given in Stein et al. [11].

We have given the equations for linear varying cells but our current implementation supports element-centered data only. The main challenge is getting the vertex program to fit within 128 lines (at the time of writing our program is 125 lines without linear support). The additional logic to support linear cells is straightforward but currently takes more than 128 instructions. The existing vertex program treats the cell as having constant values for both color **C** and extinction coefficient $\boldsymbol{\tau}$, and uses a texture map defined as [1- exp(-u)], where u is $\boldsymbol{\tau l}$, also defined in Stein [11].

4 RESULTS

Performance measurements were made on a Dell 530 with a Pentium 4 CPU running at 1.7GHz. The operating system on the machine is Linux RedHat version 7.1. The graphics card is an nVidia GeForce 4 from VisionTec. Sandia National Laboratories has many internal tetrahedral datasets but for comparative purposes we ran our performance tests on three commonly used PLOT3D datasets; the blunt fin, the liquid oxygen post and the delta wing.

Dataset	Vertices	Tetrahedra
Blunt Fin	40,960	187,395
Oxygen Post	109,744	513,375
Delta Wing	211,680	1,005,675

Table 2: Information about the three datasets tested

Dataset	Sorting	GPU	Total
Blunt Fin	.017 sec	.25 sec	.28 sec
Oxygen Post	.05 sec	.69 sec	.75 sec
Delta Wing	.09 sec	1.35 sec	1.45 sec

Table 3: Timings for the various datasets (GPU time includes loading the input registers, running the vertex program and conducting a glFinish() before stopping the timer).

* The precise color calculation of the thick vertex when the color varies linearly across the tetrahedron is beyond the scope of this paper. Please see Williams and Max [14] for the exact formula for color in this case. For our implementation we use an approximate formula given by Shirley and Tuckman [9].

The cell sorting conducted for these experiments is a standard template library sort on the cell centriods, and is not guaranteed to be correct. The timing for sorting is simply given for completeness. For production use we will use one of the advanced visibility sorting algorithms being developed in that active research area. The GPU time includes loading the input registers for each element, running the vertex program for each element and conducting a glFinish() before stopping the timer.

All cells within the dataset are rendered (we have chosen transfer functions that assign some opacity to all of the cells). Our tests were run at many different viewpoints and then averaged (the difference in timings for any given viewpoint was less than 5%). Datasets like the Blunt Fin were rendered fast enough for the viewpoint to be interactively manipulated (almost 4 frames/sec).

While the performance is reasonable, we still have room for improvement. Currently our vertex program is 125 instructions (out of the possible 128). An extremely long vertex program slows down the performance. Also most of our work has concentrated on the implementation of the PT algorithm within the GPU. At the time of writing, we believe that the biggest performance drains are the API calls and the data movement across the AGP bus. At this point we have not investigated optimization techniques for these issues.

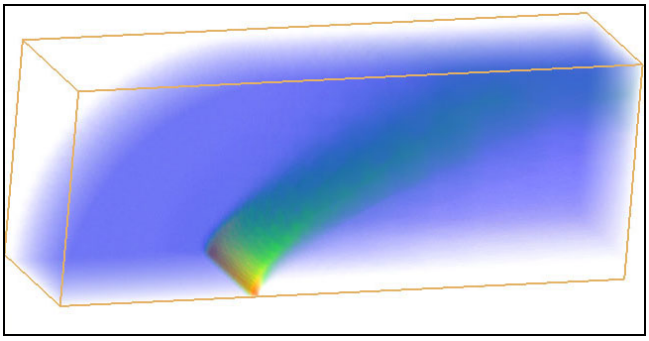

Figure 6: Image generated from the bluntfin dataset (density).

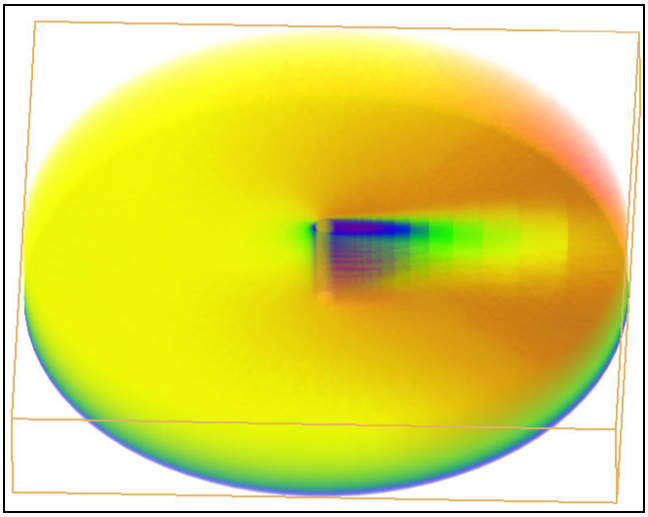

Figure 7: Image generated from the oxygen post dataset (x-momentum).

5 CONCLUSIONS AND FUTURE WORK

We have presented a method for supporting tetrahedral primitives directly on the GPU of the current generation graphics cards. Our algorithm requires no preprocessing and requires no additional data structures. We believe that techniques like ours will become increasingly advantageous as GPU's become faster and more flexible.

Our work can be extended to provide a hardware accelerated three-dimensional primitive within OpenGL (GL_TETRA_EXT). The support would involve issues like handling exponential textures internally, providing proper API calls, optimizing memory to GPU performance, and ensuring enough flexibility in the architecture to handle different use cases. The natural extension to the tetra primitive is support of tetrahedral strips and fans (see King et al. [5] for discussion of such primitives).

Currently the algorithm only supports cell-centered data (constant value across the cell). The major constraint restricting support of linear cells is the 128-instruction limit for the GPU vertex program. In the near future we will work on optimizing and reducing the code to allow for linear cell support.

We also need to fully analyze the degenerate cases where the tetrahedral points are coincident in space. Our present implementation uses a linear interpolation function for computing the thickness of the cell. The interpolation function can break down under certain degenerate circumstances and, in rare cases, produce visual artifacts.

Although the performance is reasonably fast we have clearly not tapped the full potential of the approach. As mentioned in the results section, we have not concentrated on optimizing API use and data movement across the AGP. We believe work in that area will greatly improve our performance.

6 ACKNOWLEDGMENTS

We would like to thank Claudio Silva for his excellent suggestions and advice. We would also like to thank Milt Clauser for providing our GeForce 4 card at the last minute. Funding was partially provided by the Accelerated Strategic Computing Initiative's Visual Interactive Environment for Weapons Simulations (ASCI/VIEWS) program. The DOE Mathematics, Information, and Computer Science Office also funded part of this research. The work was performed at Sandia National Laboratories. Sandia is a multi-program laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under Contract DE-AC04-94AL85000.

REFERENCES

[1] Comba, J., Klosowski, J., Max, N., Mitchell, J., Silva, C., and Williams, P. "Fast Polyhedral Cell Sorting for Interactive Rendering of Unstructured Grids," *Computer Graphics Forum*, 19: 467-478, 2000.

[2] Cook, R., Max, N., Silva, C., and Williams, P. "Efficient, Exact Visibility Ordering of Unstructured Meshes," submitted paper.

[3] Engel, K., Kraus, M. and Ertl, T. "High-Quality Pre-Integrated Volume Rendering Using Hardware-Accelerated Pixel Shading," in *Proceedings of Eurographics/SIGGRAPH Graphics Hardware 2001*, August 2001.

[4] Farias, R., Mitchell, J., and Silva, C. "ZSWEEP: An Efficient and Exact Projection Algorithm for Unstructured Volume Rendering," in *Proceedings of the Volume Visualization and Graphics Symposium 2000*, pp. 91-99, October 2000.

[5] King, D., Wittenbrink, C., and Wolters, H. "An Architecture for Interactive Tetrahedral Volume Rendering," in *Proceedings of the International Workshop on Volume Graphics*, June 2001.

[6] Max, N. "Optical Models for Direct Volume Rendering," *IEEE Transactions on Visualization and Computer Graphics*, 1(2):99-108, 1995.

[7] Max, N., Hanharan, P., and Crawfis, R. "Area and volume coherence for efficient visualization of 3d scalar functions", *Computer Graphics*, 24(5):27–33, 1990.

[8] Röttger, S., Kraus, M., and Ertl, T. "Hardware-Accelerated Volume and Isosurface Rendering Based on Cell-Projection," in *Proceedings of IEEE Visualization 2000*, pp. 109-116, October 2000.

[9] Shirley, P. and Tuchman, A. "A Polygonal Approximation to Direct Scalar Volume Rendering," *Computer Graphics: Special issue on San Diego Workshop on Volume Visualization*, 24 (5):63-70, November 1990.

[10] Silva, C., Mitchell, J., and Williams, P. "An Exact Interactive Time Visibility Ordering Algorithm for Polyhedral Cell Complexes," in *Proceedings of 1998 Symposium on Volume Visualization, pp. 87-94, 1998.*

[11] Stein, C., Becker, B., and Max, N. "Sorting and Hardware Assisted Rendering for Volume Visualization," in *Proceedings of 1994 Symposium on Volume Visualization*, pp. 83-89, October 1994.

[12] Westermann, R. and Ertl, T. "Efficiently Using Graphics Hardware in Volume Rendering Applications," in *SIGGRAPH 98 Computer Graphics Proceedings, Annual Conference Series*, pp. 169-177, July 1998.

[13] Williams, P. "Visibility Ordering Meshed Polyhedra," *ACM Transactions on Graphics*, 11(2):103-126, 1992.

[14] Williams, P. and Max N. "A Volume Density Optical Model, in 1992 *ACM Workshop on Volume Visualization*, pp. 61-68, 1992.

[15] Williams, P., Max N., and Stein C. "A High Accuracy Volume Renderer for Unstructured Data", in *IEEE Transactions on Visualization & Computer Graphics*, Vol. 4, No. 1, March 1999.

COLOR PLATE

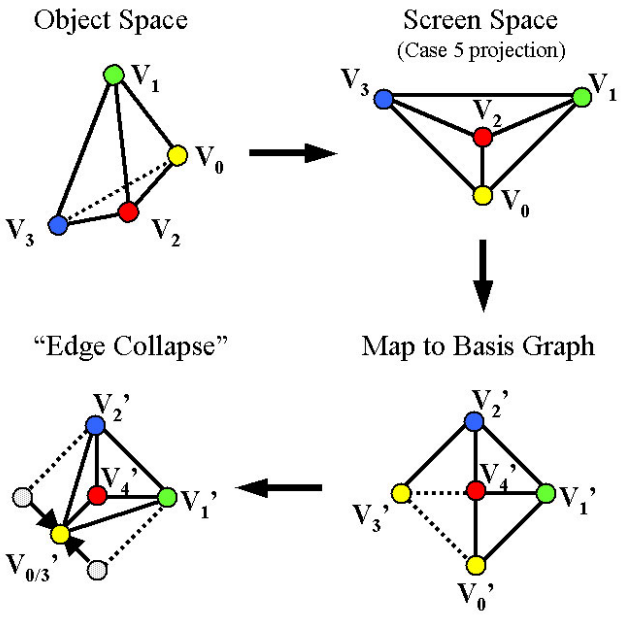

Figure 6: An example showing the vertex to node mapping from a projection instance to the basis graph (shown for a case 5 projection).

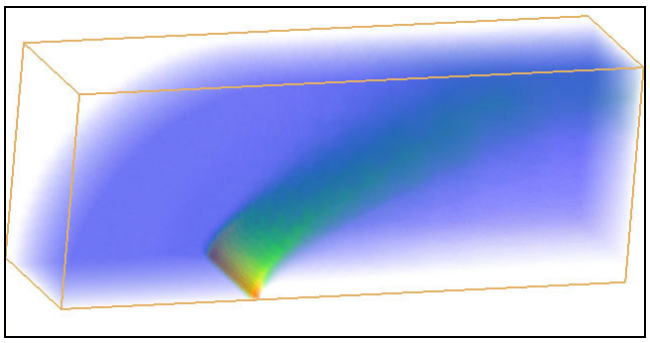

Figure 7: Image generated from the bluntfin dataset (density).

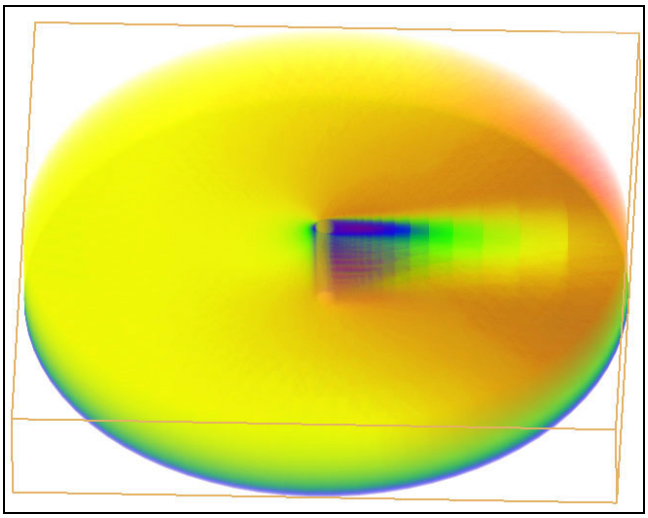

Figure 8: Image generated from the oxygen post dataset (x-momentum).